Frequently Made Mistakes in Tcl

phaseit.net/claird/comp.lang.tcl/fmm.html

Table of Contents

Introduction

Purpose of this article

This article categorizes two kinds of mistakes: evident faults, where a programmer knows he or she isn't getting the result he or she wants; and unidiomatic usages, in which functionality is at least superficially acceptable, but potentially compromised because it doesn't partake of the Tao of Tcl. In 1999, I'm also working on a section of underappreciated benefits of Tcl.
In 2001, I moved much of the material here to a variety of individual pages within the Tcl-ers' Wiki. Now, for the most part, I update the Wiki pages specific to a particular construct, and this FMM page, which aimed to span the confusions of a breadth of different commands, is mostly dormant. If you find this page helpful, I strongly urge you to give that resource--the Wiki--a bit of your attention.

In 2007, I continue to maintain this rather passively: updating it whenever someone raises a point, and occasionally adding new material, but mostly relying on the Wiki to accumulate community wisdom.

Format of this article

[Explain index. Explain format.] [Segment into Tcl- and Tk-specific?]

Testimonials

Gerry Snyder, for example, finds this "... a very valuable resource".

The mistakes

Evident faults

- #!...: Should the first line of your script look like

  > #!/usr/local/bin/wish

  or

  > #!/usr/local/bin/wish8.0

  or

  ```
  #!/bin/sh
  # the next line restarts using wish \
  exec wish "$0" "$@"
  ```

  ? Maybe none of the above. In particular, although the latter is what the distribution documentation gives, it's been known for a long time that this is in error. Tom Tromey explains why. [Explain env, ...]

- %: "My formatting doesn't work"; "I'm trying to round my numbers using 'format', but instead of geting numbers I get like ".2f", the format specifier"; "How do I format a ... in a widget?" [Explain more.] A mistake that even the experienced repeat is to write

  > bind . <Return> {.l configure -text [format "Click for %6.2f" $number]}

  when what's really wanted is

  > bind . <Return> {.l configure -text [format "Click for %%6.2f" $number]}

- ACTIVATE: listboxes take both "select" and "activate" verbs. If a listbox has focus, the active element is the one that will be select or de-selected if the user presses "space". There's rarely a need to change the default behavior regarding active elements, and most attempts to do so are confusions regarding the select verb.

- ARRAY:
  - [arrays index, but list elements are processed with lindex; this leaves ...]
  - [lots of misunderstandings about "array ..." Explain.]

- "... AUTOMATING ... my GUI ...": if you're trying to use Expectk, you're making a mistake. Read about send. [explain details]

- AWK:

> `who | awk '{ print $1 }'` works from the command line, but my Tcl interpreter rejects `exec who | awk '{ print $1 }'`

I often hear that from people who haven't yet learned to ask tclsh to interpret `exec who | awk {{ print $1 }}` Notice that the '-s aren't "about" awk; they're just conventional quoting in the shells from which one most often accesses awk. exec doesn't use the shells. {} perform an analogous function of quoting for Tcl.
More examples appear in this Google thread. Alexandre Ferrieux gives a correct and concise explanation in another thread. The problem at hand for him happened to do with `sed`, but that's essentially inconsequential; the command-line syntaxes of `awk` and `sed` exhibit identical symptoms. Briefly, as the error messages which arise themselves say, the single quotes (') are the problem. These are shell markup for strings which shall not be substituted. `/bin/sh` and others remove this markup before calling awk.

In Tcl, in contrast, braces {} serve the same purpose. Braces have the advantage that they can be nested. One conclusion: brace the braces that `awk` expects: `awk -F " " {{print $1}}`. Note that, in this last, it is **not** necessary to escape the dollar-sign, because the outer brace protect *it*, too.

Much the same has been written several times by Richard Suchenwirth, Mark Janssen, Kevin Kenny, and others; it's unlikely any particular expression was invented in isolation.

- BINDING: confusions in binding are so many and so characteristic that I might soon break them out into their own page. For now, begin by considering the my-values-are-always-out-of-date problem which

```
radiobutton .a -text a -variable variable -value a
radiobutton .b -text b -variable variable -value b

pack .a .b

bind .a <ButtonRelease-1> procedure
bind .b <ButtonRelease-1> procedure

proc procedure {} {global variable; puts $variable}

set variable blah
```

  illustrates. It typically surprises those who run this script to see that the first line which appears on selecting a button reads, "blah". Results become happier when the lines

```
bindtags .a {Radiobutton .all .a}
bindtags .b {Radiobutton .all .b}
```

  are added, as Bryan Oakley explains.
  Remember the syntax: "bind tag sequence script". It's easy to forget one of the middle elements.

- BLOCKING: [explain why AF writes "damn stupid -blocking option (so seldom useful, so frequently misused with (or instead of) fileeevents by beginners...)"]

- BUTTON: this is really about bindings and quotation (or substitution), but it invariably presents itself as a button question: someone says, "I've written

```
global i

for {set i 0} {$i < $limit} {incr i} {
      button .button$i -command {puts "This is button #$i."}
      ...
}
```

  but whichever button I push, it looks like the last button." I'll eventually explain the matter in this space; meanwhile, try

```
      button .button$i -command "puts \"This is button #$i.\""
```

  It also operates the other way; someone writes

```
button .mybutton -command "puts $my_updated_global"
```

  and is disappointed when button pushes seem to reset my_updated_global. He or she will, of course, be happier with

```
button .mybutton -command {puts $my_updated_global}
```

- C: a recurring complaint is that the C entry points for linking C and Tcl variables--Tcl_LinkVar, Tcl_GetVar, and so on--"don't work." A hint is that the complainant "put them in [his] C program", but changes "in the Tcl script" aren't properly reflected on the C side (or vice-versa). This, and a plethora of other complaints (frozen screens, un-updated variable, ...) hint at confusion about the different ways Tcl acts as "glue".

- CATCH: there are a number of confusions afoot regarding the catch command. One difficulty Chris Nelson identified is that its standard documentation is confusing. Chris has written a corrected version which accurately explains the semantics.

- CD: it's almost certain that you do not want to write

  > exec cd

  Tcl has a built-in command cd, and that's much more likely to be what you want.

- COMMAND: one imprecision most of us tolerate in our daily lives is that we use "command" in two related, but distinct senses:
    - [info commands] returns a list of commands
    - we also talk of a script made up of individual commands, when we more properly should say "command lists" (with a nod to Mark Stone's "everything in Tcl is a list") or "command strings"

  The problem is that when we look at, for example,

  ```
  set a value
  puts $value
  ```

  sometimes we say, "There are two commands, 'set' and 'puts'," and sometimes we say, "There are two command (list)s, 'set a value', and 'puts $value'." I now believe this ambiguity, or at least polysemy, is part of the source of complaints about the complexity of Tcl's syntax.

- COMMENTS: Here's the real story: '#' names a do-nothing procedure which takes a variable number of arguments. That's the best way to understand Tcl's commenting. To internalize what that means in practice often involves long, messy arguments in comp.lang.tcl which last for weeks [give a few examples of such threads]. [Explain related tangents.] Comments are detectable only where Tcl recognizes a command word: the first word after a complete command following an unquoted newline or semicolon. This tricks people, including Aeleen Frisch, in her otherwise estimable *Essential System Administration*, attempting to comment lines after a command

```
Wrong:
     set a [doCmd $b $c]     # doCmd affects global state
Need:
     set a [doCmd $b $c]     ;# doCmd affects global state
```

  Comments near cases are notorious in Tcl for causing embarrassing problems. Bryan Oakley posted particularly illuminating explanations--including one that Nir Levy labeled "the best example ever"--of the situation. Here's an abbreviation of the main idea:

```
switch $a in {
    # This comment in the obvious place causes unexpected effects
  fred {
    doFred
  }
  fred {  # this comment placement is fine if non-obvious
    doFred
  }
  default {
    doDefault
  }
}
```

  More comments on comments appear in the Wiki.

- DIR: "I'm trying to 'exec dir', and ..." Yes, it is possible to make this work; in the long run, though, you'll be much happier writing "pure Tcl" with glob.

- DOLLAR: Tcl is not Perl, and it's not C. Notice:

```
/* This is C. */
myvariable = 3;
printf("%d\n", myvariable);


# This is Perl.
$myvariable = 3;
printf $myvariable;


# This is Tcl.
set myvariable 3
puts $myvariable
```

  It is possible to write "set $var 23", but only in such a context as

```
set var myvariable
set $var 23
puts $myvariable;     # This will print "23".
```

  A related common confusion is with double de-referencing.

- DOUBLE DE-REFERENCING: In Perl, you can write `print $$myvariable;`. The analogue most Tcl beginners try doesn't give them what they want.
  As Joe Moss explains, `puts [set $myvariable]` does.

  There's more to it, though.

- ENTER: those working with bindings for the first time frequently make a mistake when they first want to program an action for the "Enter" key. They write something like

```
button .button -text Button
pack .button
bind . <Enter> {push "I hit 'Enter'."}
```

  This does *not* give the results they want, for "Enter" in the context of binding has to do with moving the cursor within the geometric boundary of a limit. It's almost certain that such programmers want

```
button .button -text Button
pack .button
bind . <Return> {push "I hit 'Enter', also called 'Return'."}
```

- ENVIRONMENT variables: [explain how they're globals ("proc a {} {glob env; puts $env(TERM)}"]

- EOF: end-of-file is marked only *after* an attempt to read beyond the end of a channel. A coding like

```
set number_of_lines $argv

set filename /tmp/example
set fp [open $filename w]
for {set i 1} {$i <= $number_of_lines} {incr i} {
        puts $fp "This is line #$i."
}
close $fp

set fp [open $filename]
while {![eof $fp]} {
        puts [gets $fp]
}
puts "Did you notice that trailing blank line?"
close $fp

set fp [open $filename]
while {-1 != [gets $fp line]} {
        puts $line
}
puts "*This* coding does what readers expect."
```

  illustrates how this frequently leads to an off-by-one error. On another hand, this fault often is inconsequential, for many programs report the same result even if (unintentionally) coded, as in this example, to receive one apparent extra blank line at the end of the file read.

- EXEC: exec doesn't use a shell. Exec is a Tcl command. Exec expects its arguments as separate words.

  > exec ls -l /bin

  does what Unix people want, but

  > set cmd "ls -l /bin"; exec $cmd

  does not, but

  > set cmd "ls -l /bin"; eval exec $cmd

  and even

  > set cmd "ls -l /bin"; exec /bin/sh -c $cmd

  do. Microsoft hostings (Windows 3.1, W95, WNT, ...) present particularly puzzling manifestations of this same:

  > exec dir

  and

  > exec \msdev\bin\nmake

  don't yield happy results, but

  > exec command.com /c dir

  (or, even better,

  > exec $env(COMSPEC) /c dir

  , or still better than that,

  > eval exec [auto_execok dir]

) and

  > exec /msdev/bin/nmake

  do. [Explain why] [See DIR.] [See long explanation of exec and relatives.] Repeat: "exec doesn't use a shell."
  Many exec solutions turn out to involve catch.

- EXPRESSION quoting:

  > expr [info exists a] ? $a : 0

  probably doesn't do what's wanted, but

  > expr {[info exists a] ? $a : 0}

  will.
  [Anyone have a punchy way to explain this?]

  Tcl 8.0 might change all this; the plan is to treat all expressions as if they are in braces, making the two examples identical.

- FIFO: we often hear complaints from people unhappy with their attempts to open Unix named pipes. Invariably, the problems have nothing to do with Tcl. Unix fifo opens by default wait for another process to open the other end of the fifo in a complementary way. As usual, Tcl gives appropriate options and access, so it's not necessary to use the default:

  > ... open $MYFIFO {RDONLY NONBLOCK}

  It's even possible to fconfigure the channel back to a blocking mode, once the open has been achieved.

- FORMAT: programmers coming from C want to use format and scan more than is necessary. Get and -text and textvar-s, for example, do most of the simple things GUI constructors want [give examples, explain more carefuly].

- GLOB: [explain] [write, "glob *.c", not "exec dir *.c". And so forth.]

- GLOBAL: "... How do I make a global variable local? ..."

```
proc my_command {} {
      global local_variable
      set local_variable something
}
my_command
puts "$local_variable is OK."
```

  This usage is *opposite* what many coming from C expect, because the `global` appears in (local) procedures, rather than "on top".
  [explain other global problems]

  The Wiki has more explanations.

- HEXADECIMAL: hex numerals must be preceded by "0x". "a" is not a hex numeral, but "0xa" is. However, "0" is a synonym for "0x0".

- IF:

  > if ($a < $b) ...

  and

  > if {a < b} ...

  are probably mistakes, but

  > if {$a < $b} ...

  probably isn't. Recall that the first argument to "if ..." is an expression to be evaluated. Another common syntax-related error that often plagues those familiar with a particular C coding style is

  ```
  if {$a < $b}
  {
        do something;
  }
  ```

  for which

  ```
  if {$a < $b} {
        do something;
  }
  ```

  or even

  ```
  if {$a < $b} \
  {
        do something;
  }
  ```

  is probably wanted.

- INTEGERs: Tcl arithmetic is not precisely defined. This most often appears as a problem in calculating large integers. Thus, for many installations,

```
expr 1234567890 --> 1234567890
expr 12345678901 --> -539222987
```

  As Dr. Ousterhout explained in Message-ID <58svee$f77@engnews2.Eng.Sun.COM>,

  > I'm not sure that this is documented anywhere, but the truth is that Tcl integers are always at least 32 bits, and sometimes more. The "sometimes more" part is a bit unpredictable because it depends on the platform and on the particular command being invoked. If the C "int" type is 64 bits on a platform then Tcl integers should always be 64 bits. However, there are some platforms (DEC Alphas?) where "int" is 32 bits but "long" is 64 bits. On these platforms expr will do its calculations using 64 bits but almost everything else in Tcl only uses 32 bits (i.e. Tcl uses the C type "int" everywhere except in expressions, where "long" is used; don't ask why).

- INTERACTIVE: "When I run my command (script, sequence, ...) interactively, inside tclsh (wish, expect, ...), it works right, but when I do it 'batch', I get (no output, an error message)." This is common, and easy to explain, although difficult to find on ones own. Tcl interpreters behave slightly differently when run interactively. This is a deliberate choice, made for the convenience of experienced users, but it certainly has the potential to confuse beginners. It is conventional for interpreters to print the value of any invoked command, say, `my_command`. To mimic the same within a batched application, simply `puts [my_command]`.
  Similarly, when a command works interactively, but reports "invalid command ..." in batch, it's probably because of the following: when an interpreter does not recognize an invoked command, it attempts to resolve it by a couple of different conventions. One is to regard the command as an external command, and invoke it through exec. Thus, for example, `ls` returns a result when run in an interactive Unix session, but can't be found as a command in batch, where it's necessary to be explicit: `exec ls`.

  There are other reasons to deprecate `exec ls`, incidentally.

- LINDEX: lindex and such other list-oriented commands must be applied to *lists*. [Explain lists vs. strings ...]

- LINKING: Linux and occasionally other Unix developers report, "I'm copying what I read in Ousterhout's book [give ref] to link together Tcl and C [give ref], but dl{open,error,sym} are undefined [give ref]. What do I do?" The answer: link with "... -ldl"

- LOST widgets: "... (eg. in canvases) when you reorder your code so that the background widget is defined after the pieces supposed to hang in front, and you forget that you then must add a 'raise' (or change the order of your definitions back)

- LREPLACE: lreplace doesn't actually replace. It returns a new list, and does *not* modify the existing one. Often people write

  > lreplace list $first $last

  when they want

  > set list [lreplace $list $first $last]

- LS: "'ls $FILENAME'" kind-of works, but 'ls *.c' doesn't give me anything sensible." Many people say this (with occasional "... 'dir $FILENAME' ..." variations). I've written elsewhere a very detailed explanation for beginners of why this is. Here's the abbreviated version:
  The '*.c' is not expanded. `ls` is trying to find a file called '*.c'. What you really want is one of the following:

  ```
  tclsh % eval ls [glob *.c]
  tclsh % eval exec ls [glob *.c]          ;# ok in scripts, too
  tclsh % exec /bin/sh -c "ls *.c"         ;# ok in scripts, too
  ```

  Unlike most UNIX shells, Tcl does NOT expand patterns such as "project*.tcl" in file names. In particular, the operation of exec is somewhat like MS-DOS's COMMAND.COM in this regard. A result is that, whereas in {sh,bash,csh,...} you'd write

  > find . "*.c" -print

  in order to ensure find sees the pattern, Tcl permits you simply to command

  > exec find *.c -print

  You often *want* pattern (sometimes called "wildcard") expansion, though. The `glob` command provides this. If you'd write

  > lpr *.c

  in shell, for Tcl you'd have

  > exec lpr [glob *.c]

  The manual page explains that [glob] generates an error if no files match unless you specify `-nocomplain`. One detail missing from the man page is that

  > [glob */]

  returns a list of subdirectories of the current directory whereas

  > [glob *]

  gives a list of files AND subdirectories.

- NO SUCH ...: this could be called the EVAL mistake, but no one realizes it's an EVAL problem until too late. If you are thinking, "I know every part of this works" or "I tried this outside and it's fine" or "sometimes it's OK, and it always works when I give it a ...", or "Why is it saying, 'no such ...'?!?", then you've probably written
    - "file delete [glob $dirname]" when you want "eval file delete [glob $dirname]"
    - "set mycommand {ap1 arg1}; exec $mycommand" when you want "set mycommand {ap1 arg1}; eval exec $mycommand"
    - ...

- NO SUCH VARIABLE: "... Why does it tell me there's no such variable when I try to use it in a procedure?" You need to understand Tcl's notion of contexts [other keywords to explain here: namespace, lexical scope, ...]. If you've scripted

```
set my_variable a_value
proc my_command {} {
    puts "The value is '$my_variable'."
}
my_command
```

  we say that `my_variable` is known only in global context, but not in the context of `my_command`. You'll be closer to what you want if you write

```
set my_variable a_value
proc my_command {} {
    global my_variable
    puts "The value is '$my_variable'."
}
my_command
```

  Warning: you're likely to come across source code that looks like

```
set my_variable a_value
proc my_command {} {
    uplevel {puts "The value is '$my_variable'."}
}
my_command
```

  uplevel is a topic you should tackle only after you're quite comfortable with global.

- PATTERN matching, including both regular expressions and glob patterns:
  forgetting to escape one or both square brackets when using them as part of a regular expression or glob sequence; thus

  > glob [A-z]*

  is BAD because Tcl tries to interpret A-z as a command, but

  > glob \[A-z]*

  is better, although still problematic if used within another command. For example,

  > lsort [glob \[A-z]*]

  matches the first [with the first closing]. What's probably wanted is

  > lsort [glob \[A-z\]*]

  or

  > lsort [glob {[A-z]*}]

- PATTERNs: "My command works when I give it a file name, but it just gives up when I pass '*' to it." This frequently puzzles developers. It's usually tied up with confusion about exec and ignorance of glob. The solution is simple: read the documentation for the latter, and substitute `[glob $wildcard]` for `$filename`.

- QUOTING: skeptics often talk about Tcl's "quoting hell". This is a reliable indication that they don't understand the simplicity of Tcl's syntax. Alexandre Ferrieux compares Tcl's parsing with that of other languages, while Bryan Oakley ...

- REGEXP: I don't know of specific problems with regexps that recur, but they're a complex subject, and certainly encompass a class of questions that arise often. The best places to start are with the man page, of course, and also Jeffrey Friedl's on-line explanations.

- SCRIPT: [explain global context]

- SED: see `awk`, which behaves identically, for these purposes.

- SPACE in filename: when developers first try to spawn a fully-qualified executable under WIN* with

  > exec /Program Files/../myprogram.exe

  they see

  > couldn't execute "\Program" ...

  What they generally want is

  > exec {/Program Files/../myprogram.exe}

  Paul Duffin has an interesting approach to letting Tcl itself figure out such quoting.

- SPECIAL characters:
  - most troublesome to beginners is the difference between "" and {} quoting which differentially allow or suppress command, variable and backslash substitution.
  - " is for string or character quoting, while () groups expressions, often in if-s and while-s.
  - "...\[..." when using a range in regular expressions
  - double-escape parentheses (and more) when trying for matches in a regular expression.
  - variable, quote, command, and backslash substitutions are performed exactly once on a word, and do not affect word boundaries.
  - in braces, the backslash char escapes don't work:

    > regexp {[^\n]+}

    looks for strings not containing backslash or n. This is symptomatic of a whole class of uneasy interactions between regexp and Tcl syntax, which many Tcl-ers cite as the single biggest headeache when writing Tcl: "what to quote?"

  See also the section on PATTERN.

- SUBSTITUTION: Section 3.10 of Tcl and the Tk Toolkit begins with a passage so accurate and apt I can only quote it in full:

  > The most common difficulty for new Tcl users is understanding when substitutions do and do not occur. A typical scenario is for a user to be surprised at the behavior of a script because a substitution didn't occur when the user expected it to happen, or a substitution occurred when it wasn't expected. However, I think that you will find Tcl's substitution mechanism to be simple and predictable if you just remember two related rules:
  > 1. Tcl parses a command and makes substitutions in a single pass from left to right. Each character is scanned exactly once.
  > 2. At most a single layer of substitution occurs for each character; the result of one substitution is not scanned for further substitutions.

  My summary: don't fight the substitution rules; spending much time worrying about substitution and escaping is usually a clue that it's time to decompose your problem with judicious definition of a new procedure or two.

- SYNTAX: here's the blunt truth: people who complain about Tcl syntax are missing something big. There is no syntax to Tcl, or almost none (mirrored here); it's most like Scheme or FORTH in that regard. What throws those coming from BASIC or C or relatives appears to be the lexification rules. Here's the entire story: a command is
  1. broken into words,
  2. substitutions are performed exactly once for each variable, quote, command, and backslash special character, ,
  3. the first word is identified as a command name, and
  4. the argument list is passed to the command for evaluation.

  A different way to say this: syntax in the grand sense includes lexification, parsing, substitution ... [explain]
  One other aspect of syntax often confuses: it's easy to introduce typographical errors--unbalanced parentheses, for example--which are only detected at run-time. This is a surprise to those accustomed to "eager" compilers.

- TUTORIAL: think you can learn Tcl/Tk from on-line documentation? You can--but you'd be making a mistake. Buy at least one of the books available, and use it up. The tutorials all make more sense with a reference at your side.

- UPDATE: your Tk application is doing what you want--probably taking some action, and showing the progress of that action--but it "waits until the end" before results appear on the screen. You might get sophisticated with background processes, pipes, fileevents, and all sorts of other hair; more likely, though, is that you just need to learn about update.

- UPVAR: upvar takes a name, not a value.

- WILD-CARDs: to conform to the nomenclature of the distribution documentation, this entry now appears under 'patterns'.

- WRONG values: "Moving code from one proc to another, forgetting to bring the 'global' statements with you so that you (with no warning!) create and access local variables rather than the globals"

- ZERO: a numeric representation (examples: "5", "639.42", ...) which begins with '0' (example: "04") is interpreted as octal. It surprises some, when, for example, "expr 09 + 1" is a syntax error. The standard approach to this common situation is such an expression as `expr [string trimleft $month 0] + 1`
  This is one of the two most common threads in common.lang.tcl (commenting is the other). The natives are sometimes grouchy on the subject.

False Tcl idioms

- Associative ARRAYs are quite wonderful. Don't avoid them by resorting to hacks such as

  > set foo_$n "value $n"; ... puts [set foo_$n]

  it's almost always better engineering to write

  > set foo($n) "value $n"; ... puts $foo($n)

  In fact, it's a good rule of thumb to be suspicious any time you see a variable name which is itself variable; that's likely to be an opportunity to engineer a more readable, better-performing, easily re-used, and idiomatic solution with judicious use of associative arrays.

- John LoVerso explains a subtlety about upvar-ed arrays in his criticism of regression in the use of nested arrays.

- BINDING: there are at least a couple of degrees of freedom in writing bindings: when
  - definition-time
  - invocation-time

  and/or in what context substitutions will be evaluated; and whether to segment the binding as a procedure. All the following are syntactically acceptable:
  - button ... -command "puts \"Some long string about $i.\""
  - button ... -command {puts "Some long string about $i."}
  - button ... -command "printMessage $i"
  - button ... -command {printMessage $i}

  but they offer different combinations of function, readability, and convenience. There are a number of reasons newcomers should be biased toward segmenting bindings with appropriate procedure definitions:
  - it improves readability--if, that is, you have a Forth-Scheme-... background. By my observation, those coming from BASIC and C sometimes are uncomfortable with one-line procedures;
  - it promotes more powerful programming [give examples of mixed evaluation-time, and quoting hell];
  - it improves performance. Sometimes. With Tcl8.0's byte-code compiler, it's generally *more* expensive to in-line bindings such as the button examples above than to "proceduralize" them. This contrasts with the performance profile of previous releases of Tcl. Finally, note in all this that I've been constructing bindings with quoting punctuation. This is common--in fact, I do it in a number of applications that are in successful operation--but a bit of a hazard. If, for example, I define a command,

    > ... puts "The 'X' binding now has the value '%X'." ...

    and %X might evaluate to a string which includes a double-quote character, it's very likely the puts won't do what I want. It's often preferable to write

    > ... puts [list The 'X' binding now has the value '%X'.] ...

    As Eric Bohlman wrote in <ebohlmanE38px1.AKJ@netcom.com>,

    > A good general rule is that if you're having substitution problems with a -command script, turning it into a proc will make things easier

    Bryan Oakley summarizes:

    > In other words, as a general rule it's good to make widget bindings call custom procs rather than hard-coding a bunch of tcl in-line. And, when doing so to use the list command to make a well-formed command.

- C-coded applications: [explain how people are OFTEN best off keeping separate processes, and they shouldn't be in such a hurry to do all the extending and linking and packaging that excites them]

- GLOBAL VARIABLES: as Ross J. Reedstrom wrote to me,

  > One programming paradigm that is useful in Tcl that took me a little while to find is using global arrays to manage the global name space, and make access to global config variables easier from deep inside procedure calls.

  [Construct good example.]

- LISTs: two positive rules about lists are paramount:
  1. use list commands (concat, lindex, list, lreplace, lsearch, ...) to build values, not string construction. What's the consequence? When we write

     > button $buttonname -command "puts $mymessage"

     , as many of us often do, we're asking for less useful handling of '$', '[', blank, and other special characters in $mymessage than

     > button $buttonname -command [list puts $mymessage]

     affords. However, there's subtlety even here, most notably in the case when we want variable substitution at both binding- and execution-time. The example then might be

     > ... -command "puts \$fid [list $mymessage]"

  2. Don't use list commands on values that are not lists, especially user data. When reading a file, for example, do not "lindex ..." to parse lines, but regexp and split your way through them. As Brent Welch advised in <59p9hp$k15@engnews2.Eng.Sun.COM>, about treating external data as lists,

     > The first unmatched double quote or brace will kill your application.

- SCAN: when a simple scan works correctly, it is generally faster than regexp processing; sometimes it's also a cleaner expression. One example of this contrasts

```
set name pt-004.tallships.i
scan $name {pt-%d.talls} n
```

with

```
regexp {pt-0*([0-9]+)\.talls} $name {} n
```

Scan is also handy for populating a collection of variables, in analogy with Perl's XXX:

> scan [clock format [clock seconds] -format {%Y %m %d}] {%d %d %d} y m d

However, scan isn't as powerful as regexps in generalizing, and sometimes those coming from C or other languages try too hard to make it work, when a simple regexp would be easier [write good example]. See also format for related information.

- SECURITY

- SHARED library: I'm working on an article on this subject, and particularly the Linux-dl-missing stuff-... issue.

- Tcl_LinkVar: the manual page prototypes the interface with "char *", and many readers misunderstand the intent for variables of TCL_LINK_STRING type. As Alexander Ferrieux and others have pointed out, it would have been better both to write the linkage as "void *", and also to explain that ...

- TRIM:

"Why does

```
set string /export/home/dickenp/mail2trace2
set strip /export/home/dickenp/
string trimleft $string $strip
```

return 'ail2trace2' return than mail2trace2?" As Mike Hoegemann writes, "the last argument to string trimleft is a *set* of characters to be trimmed NOT a suffix pattern to be trimmed." This surprises quite a few programmers, though.

- WHILE: the usual problem here is writing

  > while $var {...}

  It's almost certain that

  > while {$var} {...}

  is intended, for the first form is only an obscure way to loop for(n)ever. In the first form, $var is evaluated once, and that same value is used on each iteration. In the second form, the current value of $var is tested on each pass through the loop.

Those cultivating good Tcl style will want to read the FAQs, guides, and analyses I've collected.

Underappreciated benefits

Someone might code Tcl correctly, in the sense of being without formal errors, yet derivatively. If your Tcl source looks like Perl or Pascal, you're probably missing out on the best the language has to offer. This section explains the unique ideas that differentiate Tcl from other languages, and make for the best Tcl programming.

Simplicity

Gluing

Event model

Traces

Code-data duality

[Explain gluing--especially two-way pipes--event model (compare with IO::Multiplex), traces, and simplicity. Also emptiness, manifold technology.]

About this article

Origins

Motivations

Plans

MUCH richer cross-references; tightened writing; reader responses; ...
It's ENORMOUSLY helpful to hear from readers about whether and how this article benefits them. I'm excited about the possibility that careful cross-referencing can make this article particularly useful as a tutorial. Another live question for me now has to do with

the scale of this article; I suspect it's just a bit large to be comprehended easily, and deserves a re-write which will pare down its top-level size. Reactions, anyone?

Acknowledgements

Thanks to Andreas Kupries, Laurent Duperval, Peter A Fletcher, Pascal Bouvier, hops@sco.com (Mike Hopkirk), ketil@kvatro.no, Alex Martelli <martelli@cadlab.it>, Ross J. Reedstrom <rjr@bioc.rice.edu>, Hume Smith, and jswitzer@aimnet.com for a variety of suggestions and criticisms. Special thanks to Larry Virden, who has promoted the success of this page over and over, and Jeffrey Hobbs, who injects large volumes of de-mystification into comp.lang.tcl. Chris Nelson and Jim Graham drafted a few of the entries. Chris has also constructively criticized many of them. Jean-Claude Wippler claims this page should be glossed as "Fascinating Mistake Monitor".

Master copy

The current version of this document has a permanent home at http://phaseit.net/claird/comp.lang.tcl/fmm.html.

Index

- octal: ZERO
- return: ENTER
- scan: %, SCAN,
- select: ACTIVATE
- special characters: LISTS; SPECIAL characters
- typographical errors: SYNTAX
- zero: ZERO

Cameron Laird's notes on mistakes frequently made by newcomers to Tcl/claird@phaseit.net

=0} { puts "== $line ==" } -->